A FORMAL METHOD FOR THE ABSTRACT SPECIFICATION OF SOFTWARE

John McLean

Center for High Assurance Computer Systems
Naval Research Laboratory
Washington, D.C. 20375

An intuitive presentation of the trace method for the abstract specification of software contains sample specifications, syntactic and semantic definitions of consistency and totalness, methods for proving specifications consistent and total, and a comparison of the method with the algebraic approach to specification. This intuitive presentation is underpinned by a formal syntax, semantics, and derivation system for the method. Completeness and soundness theorems establish the correctness of the derivation system *vis−a−vis* the semantics, the coextensiveness of the syntactic definitions of consistency and totalness with their semantic counterparts, and the correctness of the proof methods presented. Areas for future research are discussed.

1. INTRODUCTION

W. Bartussek and D. L. Parnas introduced the *trace method* for the abstract specification of software in [1], at least partly, in response to Parnas' earlier observation that there is no "precisely defined notation for writing abstract specifications...that I feel to be useful" [16, p863]. The method is useful, but as presented in [1], it falls short of Parnas' goal of being "precisely defined". A formal description of the method is necessary for the proof of assertions about trace specifications and their implementations and for the design of software support for the specification user.

This report contains an intuitive presentation of the trace method, syntactic and semantic definitions of consistency and totalness,[1] methods for proving specifications consistent and total, and a comparison between the trace method and the algebraic approach to abstract specification. Following this presentation is a formal description that serves as a foundation for the method: a syntax, a semantics, and a set of inference rules for trace specifications. Soundness and completeness theorems establish the correctness of the rules of inference *vis-a-vis* the semantics, the coextensiveness of syntactic definitions of consistency and totalness with their semantic counterparts, and the correctness of the proof methods presented. A concluding section discusses areas for future research.

Strictly speaking, the report is self-contained with respect to both formal logic and the trace method. Nevertheless, some background in logic, as, e. g., can be obtained from [11], would be useful, as would an informal understanding of the trace method, as presented in [1]. An elementary knowledge of set theory, as, e. g., given in [14], is assumed.

2. AN INTUITIVE PRESENTATION OF TRACE SPECIFICATION

This section contains an intuitive presentation of the trace method that serves as an introduction to and a motivation for the formal description that follows. An informal discussion of the syntax and semantics for the method precedes discussions of consistency and totalness and a comparison of the method with the algebraic method of specification. Although the correctness of the techniques described here for reasoning about a specification presupposes the foundation that follows, the techniques can be understood and used without a knowledge of their formal underpinnings.

2.1. Syntax and Semantics

The trace method is a formal method for the abstract specification of software, where "software" is liberally construed to cover any program (procedure) or set of programs. As such, in so far as the terms "abstract data type" and "module" are used to refer to sets of related programs, it is a method for their

1. I use *total* in the same sense that Guttag uses *sufficiently-complete* in [5]. The reason for the change in terminology is that *completeness* already has (several) established senses in logic circles. Giving it one more simply breeds confusion.

specification as well.

Being formal does not distinguish the trace method from many other methods for specifying software. However, being abstract does. A trace specification describes only those features of a program that are observable; it specifies what the program does without describing an algorithm for doing it. If the use of a particular algorithm is required, then this is included as a constraint to, not as a part of, the specification.[2] In this respect, it differs from procedure specification methods that are based on "operational definitions" and abstract data type specification methods that are based on "abstract models" [10]. For our purposes, the most important feature of these latter classes of methods is that they specify software by giving a paradigm implementation.

Consider a module that takes an integer array as input and returns the index of a maximal element in the array. A trace specification states only that the return value indexes an array element that is greater than or equal to any other element in the array. It leaves the method for finding this value completely open. In contrast to this, both the operational definition and the abstract model specification methods specify the module by giving a paradigm implementation such as the one below.

```
$x$ :=1
for $i$ =2 to $size(array)$
    if $array(i) > array(x)$ then $x := i$
return($x$)
```

The trace specification has two advantages over the latter.[3] First, the former specification does not contain clutter; it contains neither artifacts from a particular language for presenting algorithms nor artifacts from a particular algorithm. This makes it more perspicuous and easier to handle in proofs about implementations. More importantly, by not having artifacts from a particular algorithm, it is free from a whole class of potential misunderstandings: those that result from the attempted gleaning of its essential features from a mass of extraneous details. Although the latter specification may be intended to allow any program that is functionally equivalent to the one given, it is not clear what *functionally equivalent* means here. For example, when there is more than one maximal element in an array, it is not clear from the latter specification whether the programmer is required to return the smallest number that indexes a maximal element or not. This point can be clearly seen by noting that if the program *were* required to return the smallest index, then the specification of this new program would be identical to the specification of the original. The trace specification, however, would contain a new assertion that reflects the additional requirement.

Trace specifications are also conducive to good programming practice. By requiring program output to be specified solely in terms of input, the trace method not only forces designers to make any information shared by two or more modules part of an explicit interface, it also discourages unnecessary modular coupling by focusing the designer’s attention on such shared information. This makes independent implementation of modules possible and leads to understandable software that is easier to maintain [15].

In the trace specification method, programs are specified by describing three properties they possess:

(1) What do the access procedures of the programs look like, i. e., what are their names, their parameter types, and their return values types if any? These properties are given by sentences of the form $proc : parameter_1\ type\ \ldots\ parameter_n\ type$ --> $return\ value\ type$.

(2) Which sequences of procedure calls (called *traces*) are *legal*, i. e., are not regarded as being in error? These are given by assertions of the form $L(trace)$.

(3) What is the output of legal traces that end in a function call? This value is denoted by $V(trace)$.

To make specifications more readable, we introduce a relational predicate into the language. If two traces agree on (1) current legality, and (2) legality and return value for future program behavior, then we say they are *equivalent* and write $trace_1 \equiv trace_2$.

2. See [6,7] for a discussion of the problem of presenting algorithmic constraints in an abstract specification or requirements document.

3. See [16] for a general discussion of the advantages of abstract specification.

Note that in giving the parameter types of a procedure call, we need not limit the programmer unnecessarily since the parameter types can be as abstract as we please. For example, we may specify the parameter type of a procedure as *set* early in the design phase and replace this parameter type by *vector* if we choose to use a vector representation for sets. We can later replace the latter parameter by *array* if we desire. Similarly, we are not restricted to a fixed number of parameters for a given procedure. As originally described [1, 12], the trace method followed other methods for software specification in demanding a rigid syntax for procedure calls. Each procedure of a specified module had to take a specified number of parameters of a fixed type. However, this requirement does not fit in well with the current state of programming language research and was strained by built-in procedures even of traditional languages, e. g., PL/1 stream edited I/O procedures which take an arbitrary number of arguments. The trace method as described in this paper allows for a more flexible syntax for procedure calls by including schemata.

As an example, consider the following specification of a stack module that permits multiple pushes and pops. The module contains three procedures: *PUSH* takes an arbitrary positive number of some, as of yet, undetermined type of object as parameters but returns no value; *POP* either takes either no parameter or a positive integer as a parameter, but returns no value; and *TOP* takes no parameters but returns an object. The syntax of the module is specified thus.

$PUSH : obj \ \cdots \ obj$

$POP : [int]$

$TOP : \text{-->} \ obj$

The syntax for *PUSH* is specified by a schema that represents an infinite number of sentences: viz. $PUSH : obj$, $PUSH : obj \ obj$, etc. Each one of these sentences describes a legal call on *PUSH*. The programmer is free to treat *PUSH* as a single procedure that takes an unspecified number of parameters or as a set of procedures, each of which takes a different number of parameters from the rest. Similarly, *POP* is specified by a schema that represents two sentences: $POP :$ and $POP : int$.

The semantics of the module consists of eight assertions describing the module's behavior: (1) if a series of procedure calls has not resulted in an error, then *PUSH* can be legally called with any object parameter; (2) calling *TOP* does not result in an error if and only if calling *POP* does not; (3) calling *PUSH* with multiple parameters is equivalent to calling *PUSH* with each parameter individually, leftmost first; (4) $POP(1)$ is equivalent to *POP*; (5) calling *POP* with n as a parameter is equivalent to calling *POP* n times; (6) calling *PUSH* followed by *POP* does not affect the future behavior of the module; (7) if *TOP* can be legally called, then calling it does not affect the future behavior of the module; and (8) the value of any legal series of procedure calls ending in *PUSH* followed by *TOP* is the parameter of the last *PUSH*. These assertions are symbolized as axioms in an extension of predicate calculus.

(1) $L(T) \text{ --> } L(T.PUSH(o))$

(2) $L(T.TOP) \text{ <--> } L(T.POP)$

(3) $T.PUSH(o_1, \ldots, o_n) \equiv T.PUSH(o_1).PUSH(o_2, \ldots, o_n)$

(4) $T.POP \equiv T.POP(1)$

(5) $i > 1 \text{ --> } T.POP(i) \equiv T.POP.POP(i-1)$

(6) $T \equiv T.PUSH(o).POP$

(7) $L(T.TOP) \text{ --> } T \equiv T.TOP$

(8) $L(T) \text{ --> } V(T.PUSH(o).TOP) = o$

Note the use of --> and <--> for *if then* and *if and only if* respectively. The connectives $\neg$, $\&$, and $\vee$ are also allowed for *not, and,* and *or*, respectively, as well as the existential quantifier $(\exists\alpha)$ for *there exists* α and the equality symbol, =. The *less than* symbol, >, is assumed to be previously defined. The variable i is assumed to range over integers, the variables $o, o_1, o_2, \ldots$ over objects, and the variable T over traces (including the *empty trace*, e, which denotes the null sequence of procedure calls). Any occurrence of a variable that is not within the scope of an existential quantifier is assumed to be universally quantified. Hence, for example, the first assertion says that for any trace T and for any object o, if T is legal then so is T followed by $PUSH(o)$. Note that the dot (.) concatenates procedure calls. Assertion (3) is a schema that tells how to reduce each of the possible calls involving PUSH that contains more than one parameter. As such, it really makes an infinite number of assertions, each involving a finite number of parameters. A formal presentation of the language is contained in the section of this paper on formal syntax.

As an example of how to use the specification to make inferences about an implementation's behavior, consider the trace *PUSH*(*tom*,*jerry*).*POP*.*TOP*. Substituting the empty trace for *T* in assertion (3) allows us to conclude that *PUSH*(*tom*,*jerry*).*POP*.*TOP* is equivalent to the sequence *PUSH*(*tom*).*PUSH*(*jerry*).*POP*.*TOP*. Using assertion (6), we can conclude that *PUSH*(*tom*).*PUSH*(*jerry*).*POP* is equivalent to *PUSH*(*tom*), and hence, that the original trace is equivalent to *PUSH*(*tom*).*TOP*. Using assertion (8) and the assumption that the empty trace is always legal, we can conclude that the original trace returns the value *tom*. A formal derivation system for reasoning about a module's behavior is contained in the section of this paper on formal deduction.

2.2. Consistency

To say that a specification is *consistent* is, intuitively, to say that it does not contain a logical contradiction. The problem with an inconsistent specification is that since anything follows from a contradiction, no implementation could do all that the specification requires. The specification places too many constraints for an implementation to be possible.

There are two ways of formally explicating this intuitive notion of consistency. One is to call a specification consistent if and only if a contradiction cannot be derived from it. This definition is totally syntactic in that it does not depend on the meaning of the symbols used in a specification, but only on the formal definition of *derivation* given in the section on deduction. Hence, we call a specification that is consistent in this sense *syntactically consistent*. The second way is to call a specification consistent if and only if it has a *model*. This definition is semantic in that one must give the meanings of the symbols involved in the specification language to state formally what it is to be an acceptable model. This is done in the section of this paper on formal semantics, and the resulting explication of consistency is called *semantic consistency*.

Each definition suggests its own method for proving consistency: showing that a contradiction cannot be derived, for syntactic consistency, and showing that a model exists, for semantic consistency. Further, each of these methods has its own set of circumstances in which it is better than the other. This stems from the fact that *ceteris paribus,* it is more enlightening to demonstrate something constructively than nonconstructively, and the fact that a constructive proof is easier to evaluate. If I give a model for a specification, there can be no doubt that the specification is consistent, and further, I gain information about the specification. It is not nearly so easy to convincingly establish directly that no derivation of a contradiction is possible, and even if I could, I gain little information about the specification. Hence, model building is the standard method for proving consistency. On the other hand, the primary method for proving a specification inconsistent is to derive a contradiction from it. The reason is again that deriving a contradiction from a specification establishes inconsistency much more clearly and gives much more insight than an argument that no model exists. Hence, it is desirable to be able to work with both definitions.

This raises a technical point. If we are to have the benefit of two concepts of consistency, we must establish that the two concepts are coextensive, i. e., that a specification is syntactically consistent if and only if it is semantically consistent. Although the coextensiveness of these two concepts (and their corresponding methods for proving consistency) is implicitly assumed by virtually all writers in the area of software specification, it must be established for each specification technique. Giving a model establishes the impossibility of deriving a contradiction (and *vice versa*) only if the model and the derivation system correspond. That the model and derivation system given in this paper correspond is proven from two theorems: a *soundness* theorem shows that an assertion is derivable only if it is true in all models, and a *completeness* theorem shows that an assertion is derivable if it is true in all models. The remainder of our discussion of consistency in this section consists of intuitive applications of these two theorems.

Consider the following specification for a module that manipulates multiple integer stacks. As before, we have the procedures *PUSH*, *POP*, and *TOP*, but here, each one operates on the stack whose name is passed to to the procedure as a parameter. This module also has a new procedure *DEPTH* which returns the number of elements in the named stack. Number theory is assumed, as is the predefined data type *name*, consisting of the set of finite character strings. The variables a and b are assumed to be integers, and r and s are assumed to be names.

Syntax:

$PUSH: int\ name$

$POP: name$

$TOP: name \dashrightarrow int$

$DEPTH: name \dashrightarrow int$

Semantics:

(1) $L(T) \text{-->} L(T.PUSH(a,s))$

(2) $L(T.TOP(s)) \text{<-->} L(T.POP(s))$

(3) $T.DEPTH(s) \equiv T$

(4) $T.PUSH(a,s).POP(s) \equiv T$

(5) $r \neq s \text{ --> } T.PUSH(a,s).PUSH(b,r) \equiv T.PUSH(b,r).PUSH(a,s)$

(6) $L(T.TOP(s)) \text{ --> } T.TOP(s) \equiv T$

(7) $L(T) \text{ --> } V(T.PUSH(a,s).TOP(s))=a$

(8) $L(T) \text{ --> } V(T.PUSH(a,s).DEPTH(s))=V(T.DEPTH(s))+1$

(9) $(L(T)\ \&\ r \neq s) \text{ --> } V(T.PUSH(a,s).DEPTH(r))=V(T.DEPTH(r))$

(10) $V(DEPTH(s))=0$

This specification is simpler than the previous stack specification in not allowing multiple pushes or pops, but it is more complicated in that it contains the new assertions (3), (5), (8), (9), and (10). Assertion (3) tells us that calling $DEPTH$ does not affect the stack, and assertion (5) tells us that if we are pushing integers on different stacks, then the order of the pushes is irrelevant. Assertions (8)-(10) tell us how to compute the value of traces ending in $DEPTH$.

The condition that $r \neq s$ is needed in the antecedent of assertions (5) and (9). If, for example, assertion (5) were replaced by the simpler assertion $T.PUSH(a,s).PUSH(b,r) \equiv T.PUSH(b,r).PUSH(a,s)$, we could derive a contradiction from the specification, given number theory, as follows. The new assertion would allow us to conclude that $PUSH(3,st).PUSH(5,st)$ is equivalent to $PUSH(5,st).PUSH(3,st)$. But by assertion (7), $PUSH(3,st).PUSH(5,st).TOP(st)$ returns the value 5 and $PUSH(5,st).PUSH(3,st).TOP(st)$ returns the value 3. From the equivalence between the two traces, we can derive 5=3, and from number theory, we can derive $5 \neq 3$. Hence, we can derive the contradiction $5=3\ \&\ 5 \neq 3$.

Proving that the original specification has a model is a bit more difficult. Intuitively, a model for a trace specification consists of a tuple of domains D and an interpretation function I that assigns to each meaningful element of the trace language an element in one of D's domains. Here, D consists of three domains: $TRACE$, the set of (possibly null) character strings that form sequences of procedure calls, contains denotations for traces; INT, the set of integers, contains denotations for integer constants; and $NAME$, the set of finite character strings, contains denotations for name constants. Hence, $TRACE$ and $NAME$ both consist of character strings. A string such as "PUSH(6,st)TOPPOPDEPTH" can appear in either, although it probably serves as a denotation only in the former.

The interpretation function, I, assigns to each variable-free trace the corresponding character string in $TRACE$, to each integer constant the corresponding integer in INT, and to each name constant the corresponding character string in $NAME$. For example, I assigns to $DEPTH(st).TOP$, 5, and $stack1$, the denotations "DEPTH(st)TOP", the integer 5, and "stack1" respectively. The denotation of language element ϕ, is written $I[\phi]$. Hence, in the model we are constructing now, $I[stack1]$="stack1".

To translate assertions in the specification into assertions about domain objects, we must also fix the denotations of relation symbols and function symbols. Let ι range over integers and σ over character strings. $I[+]$ is the addition function and $I[=]$ is the equality predicate. $I[L]$ is the predicate over strings in $TRACE$ that is true of a string x if and only if for any $s \in NAME$ and any substring ϕ of the form

"POP(s)" or "TOP(s)", there are more substrings to the left of ϕ in x of the form "PUSH(ι,s)" than of the form "POP(s)". $I[V]$ is a function f from those strings in $TRACE$ of which the predicate $I[L]$ is true and that end in "TOP(σ)" or "DEPTH(σ)", to the integers. For every string x in the domain of f, $f(x)=n$ if and only if there is a $s \in NAME$ such that (1) x ends in "DEPTH(s)", and n is the number of substrings of the form "PUSH(ι,s)" in x minus the number of substrings of the form "POP(s)" in x, or (2) x ends in "TOP(s)" and scanning x from right to left, the substring "PUSH(n,s)" is the first occurrence of a substring of the form "PUSH(ι,s)" in x that cannot be paired with a previous, unpaired substring of the form "POP(s)". Note that we do not have to assign a denotation to the equivalence symbol since it is a defined term.

The last step in giving the model is to define what it means for any assertion to be true. An assertion containing no connectives or variables is true if and only if the corresponding assertion in the model is true. An assertion containing variables is true if and only if the corresponding assertion is true for every object in the appropriate domain. An assertion containing Boolean connectives is true if and only if the Boolean combination of the corresponding element assertions is true.

Although the interpretation just given is trivial, it can serve as a model for an implementation that works, not in the standard way of mimicking stacks with arrays, but by forming a character string that represents the history of procedure calls made to the module, and using this string to calculate all needed return values. Hence, it gives us a new tool for reasoning about the module's behavior. If we wrongly assumed that DEPTH returned, not the present depth of the stack, but rather the maximum depth that the stack had attained in its history, we would discover our mistake. If we had proven the specification consistent by syntactic means, however, this fact never would have come to light. Further, such implementations can be constructed mechanically. Hence, they can serve as a basis for rapid prototyping as described in [3].

All models contained in this paper are similar to the one just given in that they regard an implementation as a transform on a string of procedure calls. This is a useful way to view software when reasoning about module behavior, specification consistency, and as we shall see in the next section, specification totalness. The reader who has no understanding of the formal underpinnings of this approach, can nevertheless use the technique to great benefit.

2.3. Totalness

An inconsistent specification fails by placing too many constraints on its intended implementation. A specification can also fail by placing too few constraints, leaving the programmer in the air about module behavior. A specification is *total* if it asserts what its implementation should do for all legal input. For an abstract specification, this is the same as asserting what output should be generated by each legal string of procedure calls that ends in a function call.

As for consistency, there is a syntactic and a semantic formalization of this property. A trace specification is *syntactically total* if for every variable-free trace T that ends in a function call, we can derive $L(T)$ only if there is some constant a such that we can derive $V(T)=a$. That is, if such a trace is legal, then it returns a specified value. A trace specification is *semantically total* if for every variable-free trace T such that $L(T)$ is true in all models, there is a constant a such that $V(T)=a$ is true in all models.[4] That is, all implementations of a *semantically total* specification are indistinguishable as far as observable behavior is concerned.

As for consistency, it is convenient to have two different concepts of totalness. The semantic concept is easier to work with than the syntactic concept in proving that a specification is not total since it is easier to show that there are two models that disagree on what to assign $V(T)$ than it is to show that there is no derivation of the form $V(T)=a$. However, the syntactic concept is generally easier to deal with in proving totalness. We must, of course, prove that the two concepts are coextensive. As for consistency, their coextensivenes follows from the soundness and completeness theorems.

4. Note that the two definitions of *totalness* bring with them syntactic and semantic definitions of *legality*. A trace T is *syntactically legal* if $L(T)$ is derivable; it is *semantically legal* if $L(T)$ is true in all models. The completeness and soundness theorems establish that a trace is syntactically legal if and only if it is semantically legal.

In this section we will consider a semantic proof that the following keysort specification, adapted from a similar one suggested by Parnas, is not total.[5] The module stores an unbounded number of ordered pairs of integers, which are inserted by the procedure *INSERT*. The first member of each ordered pair acts as a key for the procedures *FRONT*, which returns (without removing) a pair with a minimal key, and *REMOVE*, which removes a pair with a minimal key. The data type *pair* is the set of integer ordered pairs, i. e, $pair = \{(x,y): x \in int \ \& \ y \in int\}$.

Syntax:

INSERT: *int int*

REMOVE:

FRONT: --> *pair*

Semantics:

(1) $L(T) \dashrightarrow L(T.INSERT(a,b).REMOVE)$

(2) $L(T.FRONT) <\!\!-\!\!-\!\!> L(T.REMOVE)$

(3) $L(T.FRONT) \dashrightarrow T.FRONT \equiv T$

(4) $V(T.INSERT(a,b).FRONT)=(a,b) \dashrightarrow$
$(T.INSERT(a,b).REMOVE \equiv T \ \vee$
$(V(T.FRONT)=(a,b) \ \& \ T.INSERT(a,b).REMOVE \equiv T.REMOVE.INSERT(a,b)))$

(5) $V(T.INSERT(a,b).FRONT) \neq (a,b) \dashrightarrow$
$T.INSERT(a,b).REMOVE \equiv T.REMOVE.INSERT(a,b)$

(6) $V(INSERT(a,b).FRONT)=(a,b)$

(7) $(V(T.FRONT)=(a,b) \ \& \ V(T.INSERT(a^*,b^*).FRONT)=(x,y)) \dashrightarrow$
$((a<a^* \ \& \ x=a \ \& \ y=b) \ \vee$
$(a>a^* \ \& \ x=a^* \ \& \ y=b^*) \ \vee$
$(a=a^* \ \& \ x=a \ \& \ (y=b \ \vee y=b^*)))$

Assertion (7) requires that *FRONT* returns the pair with the smallest key if there is one, or a pair with that key if there is more than one. As such, the behavior of the module is nondeterministic when there is no unique pair with the smallest key. Assertion (4) requires that *REMOVE* removes the pair that *FRONT* would return if it were applied to the same trace.

We show that the specification is not total by giving two models, M and M*, and showing that there is a legal, variable-free trace T ending in a function call and a constant a such that $V(T)=a$ in M, and $V(T) \neq a$ in M*. We will use x and y as integer variables in describing the model.

As in the stack example, each model consists of an interpretation function I from the specification language to a tuple of domains D, whose elements, *TRACE* and *INT*, are as before. M and M* also agree on assigning to L the predicate that is true of all and only those strings in *TRACE* such that to the left of every "REMOVE" or "FRONT" in the string there are more strings of the form "INSERT(x,y)" than of the form "REMOVE".

For M, $I[V]$ depends on the normalizing algorithm defined below. It takes as input strings of procedure calls and deletes "INSERT(x,y)"-"REMOVE" pairs where the insert is the rightmost occurrence of an "INSERT(x,y)" that is both to the left of the "REMOVE" and has a minimum key. The parameter pair of the last "INSERT" deleted is returned.

```
NORMAL(string) RETURNS(pair):
1.    If $I[L]$ is not true of string, then abort.
2.    Remove each occurrence of "FRONT" from string.
3.    While string contains at least one "REMOVE" do
```

5. It should be noted that the specification is not total in order to make the module's actions nondeterministic, as described below, when there is no smallest key.

(a) Find the leftmost string of the form "REMOVE" in *string*, and assign to A the string up to (but not including) this "REMOVE" and to B the string following (but not including) this "REMOVE".[6]

(b) Assign to *min* the smallest key contained in a string of the form "INSERT(x,y)" that occurs in A.

(c) Delete the rightmost "INSERT(x,y)" from A such that $x = min$, and assign to *pair* the parameter pair of that "INSERT".

(d) Assign to *string* the concatenation of A with B.

4. RETURN(*pair*)

$I[V]$ is a function f from those strings that $I[L]$ is true of and that end in "FRONT" to ordered pairs of integers (x,y). $f(string)=(x,y)$ if and only if when after the rightmost "FRONT" of *string* has been replaced by "REMOVE", NORMAL(*string*) returns (x,y).

For M*, we use the algorithm NORMAL*, which is exactly like NORMAL except that the word *rightmost* in step (c) of statement (3) is changed to *leftmost*. $I^*[V]$ is defined exactly like $I[V]$ except that NORMAL* is used instead of NORMAL. To see that the keysort specification is not total, one must merely note that if we consider $T = INSERT(1,5).INSERT(1,6).FRONT$ then $V(T)=(1,6)$ in M and $V(T)=(1,5)$ in M*.

For proving specifications total, there is a standard semantic approach that suggests itself. Call a specification *theory−complete* if for every assertion A in the specification language, either A or $\neg A$ is derivable. If we can show that a specification is theory-complete and give one model in which for every legal, variable-free trace T ending in a function call, $V(T)=a$ is true for some constant a, then it follows that the specification is total. What is appealing about this approach is that there are standard methods for proving theory-completeness [2]. However, the approach is limited in that no specification containing first order number theory is complete in this sense [4].

Nevertheless, there is a straight forward method to prove a specification total. Call the number of procedure calls contained in a variable-free trace the *length* of the trace. We prove a specification total if we establish by induction on trace length that for any legal, variable-free trace T ending in a function call, there is a constant a such that $V(T)=a$ is derivable. As an example, I will use this approach to prove the keysort specification total when we replace assertion (7) of the specification by the following:

(7*) $(V(T.FRONT)=(a,b) \ \& \ V(T.INSERT(a^*,b^*).FRONT)=(x,y))$ -->
$((a<a^* \ \& \ x=a \ \& \ y=b) \vee (a \geq a^* \ \& \ x=a^* \ \& \ y=b^*))$

Note that for any trace T of the form $INSERT(x_1,y_1). \cdots .INSERT(x_n,y_n).FRONT$, where each x_i and y_i is an integer constant, there is a derivation of $V(T)=a$ for some ordered-pair constant a using axioms (6) and (7*).[7] Hence, totalness follows if we prove that each variable-free, legal trace is equivalent to a variable-free sequence of *INSERT*'s.

Assuming that any variable-free trace S of length less than n such that $L(S)$ is derivable is equivalent to a variable-free sequence of *INSERT*'s, we prove that any such trace T of length n is equivalent to a variable-free sequence of *INSERT*'s. We assume that T is of the form $S.C$ where S is a sequence of variable-free *INSERT*'s and C is a procedure call. C can be of 3 possible forms.

(1) IF C is *INSERT*, we are done.

(2) If C is *FRONT*, then by assertion (3) T is legal only if $T \equiv S$. Since S is of the required form by hypothesis, we are done.

6. Note that B may be assigned the null string.

7. It may not be clear how to apply assertion (7*). Our deductive system contain axioms that allow us to infer that an initial segment of any legal trace is legal and that there is a value for any legal trace that ends in a function call. Since the empty trace is legal, we can use assertions (1) and (3) to prove that any sequence of *INSERT*'s followed by a *FRONT* is legal. Hence, we can always infer that there is a value for such a sequence. Proving totalness consists in showing that there is a *constant value* that is derivable for a legal trace that ends in a function call if the trace is variable-free.

(3) IF C is *REMOVE*, then by assertion (2), T is legal only if *S.FRONT* is legal. Further, S cannot be empty.[8] Using assertions (4) and (5) we can eliminate the *REMOVE*.

2.4. Comparison with the Algebraic Approach

The trace method has much in common with the algebraic approach to specification, as epitomized, for example, in [5]. They are both methods of abstract specification and therefore, seem much alike when compared to such alternatives as the operational definition and abstract model approaches discussed earlier. Further, the generality of the term *algebraic* allows the development of algebraic models that are formally equivalent to the trace method.[9]

Nevertheless, the reader will notice two differences between the trace method for specifying software modules and the algebraic approach. First, the so-called *type of interest* is not necessary in a trace specification. For example, within the stack specification, the word *stack* is never used. As such, the specification corresponds more closely to how the user actually sees a stack module, *viz*. a set of access procedures with certain properties, than the algebraic method, which gives relations between the possible *values* stacks can assume. Treating stacks as values renders it necessary, even in the single stack case, to regard each procedure as taking a stack as a parameter and any procedure that affects the *inner state* of the module (called $O-functions$ in [1]) as returning a stack. There is certainly no reason for the procedures in an implementation of the module to contain such an abundance of parameters and return values, yet if the user is given the freedom to leave certain parameters out of an implementation, we lose the advantages of abstract specification discussed earlier.

In trace specification designers are free to use parameters of an abstract type *stack* if they desire, but they are also free to replace this type by a concrete representation if necessary. This is desirable since most real-world programming languages do not allow for the free creation of new data types. However, designers using algebraic specifications cannot bind their abstract data objects to a particular representation. Hence, the interface is ambiguous in that the programmer must decide whether to treat the parameter as a name or as one of several possible data objects, e. g., an array. Choosing to represent the parameter as a data object would be particularly bad since it would force the programmer to represent each stack as a separate data object, ruling out implementations of the type described in this paper or that use, e. g., a single array that stores both names and integers. The programmer faces similar problems in dealing with the artificial error values the algebraic approach needs for its stacks to assume given bad input and the unnecessary functions it needs to start, i. e., map an empty value space to an initial stack.

A slightly different problem that results from treating stacks as values is that it obliterates the distinction between a function call and the value returned by that call. This renders it impossible to represent a sequence such as $call_1.call_2$ except by treating $call_1$ as a parameter of $call_2$.[10] This is not only unintuitive for many implementations, it makes it impossible to represent sequences such as $PUSH(i,s).TOP(s).TOP(s)$ since the first occurrence of *TOP* returns an integer while the second occurrence needs a stack for a parameter.

The second difference between the two methods concerns the languages involved. The trace method makes free use of first order logic with identity while most algebraists prefer a more restrictive languages that does not contain an existential quantifier. As such, the trace language has more expressive power. An example occurs in the formalization that follows where one axiom of the deductive system states that any legal trace expression ending in a function call must return some value, without saying what that value is. This axiom cannot be written without an existential quantifier, and hence, is unavailable to the algebraist. Yet the axiom is necessary for specifying, e. g., an integer generating module whose only restriction is that it returns a different integer each time it is called. The trace specification for such a module is simple.

---

8. By the soundness theorem a trace is syntactically legal only if it is semantically legal. Since the string *REMOVE* is not legal in *M*, it is not semantically legal.

9. In particular, the trace method is formalized in first order logic, which is a cylindric algebra [9].

10. On the other side of the coin, it should be noted that the trace method makes a sharper distinction between function calls and the values they return than do most programming languages. As such, the trace method cannot represent calls that take as a parameter the return value of another call as naturally as the algebraic method can.

Syntax:

GEN: --> *int*

Semantics:

(1) $L(T)$

(2) $S \neq e$ --> $V(R.S) \neq V(R)$.

Readers should find it enlightening to try to specify this same module algebraically since they will run into problems, not only in trying to capture the nondeterminism of the module, but also as discussed above, in trying to represent sequences of function calls. Although the richness of the trace language implies that consistency and totalness is harder to establish with the trace method, nobody has found a sufficiently rich language for which consistency and totalness are decidable. Further, the methods employed in this paper constitute an important step in developing a uniform method for establishing trace specifications consistent and total.

3. A FORMAL FOUNDATION FOR TRACE SPECIFICATION

We must provide a formal foundation to support the intuitive presentation given above if we are to have confidence in the methods presented there. This foundation consists of a formal syntax, a formal semantics, and a formal derivation system for trace specifications. A soundness theorem and a completeness theorem establish the correctness of the derivation system *vis–a–vis* the semantics.

3.1. A Formal Syntax

The first step in formalizing the notion of a trace specification is to precisely define the term *trace specification*. Such a definition consists of giving a specification language **L** and stating how the well-formed expressions of **L** can be combined to yield a specification. Although some of the specifications given in the previous section contain constructs that are, strictly speaking, not well-formed with respect to the language given here, they can easily be converted to well-formed specifications.[11]

3.1.1. Language for Specifications

L is defined by giving its vocabulary and the formation rules that combine vocabulary elements into well-formed expressions. It is the smallest set that both contains its vocabulary and is closed under its formation rules. Since the syntax of the semantic section of a specification depends on the procedure-call descriptions given in the specification's syntax section, the formation rules for the former are given in terms of the latter. We assume that certain well-specified, countable, nonempty domains are given.

3.1.2. Vocabulary

The vocabulary of **L** contains that of first order logic. This is necessary to make boolean assertions about traces. As such, it contains parentheses (,); the logical connectives ¬, &, v, -->, <-->; the existential quantifier $\exists$; and the equality symbol =. **L** also contains ten vocabulary element types indigenous to traces. The first three allow for the construction of traces, the next two allow for assertions about traces, and the remaining element types allow for statements about parameter domains. These ten vocabulary types follow.

Trace Variables:

> A, B, C, ... are each a *trace variable*. They can be superscripted. They are used to make general assertions about traces.

Empty Trace:

> e is the *empty trace*. It denotes the null string of procedure calls.

11. The point is that although the formal syntax is a necessary underpinning of the method, one can relax the syntax of the language to make specifications easier to write as long as the specification can be written in the strict language if necessary. See also the discussion of schemata that concludes this section of the paper.

Procedure Names:

Any finite character string is a *procedure name*. When composed with the appropriate parameters, it forms a procedure call. In the stack example, *PUSH*, *POP*, and *TOP* are procedure names.

Trace Predicate:

The unary predicate L is the only *trace predicate*. As stated in the previous section, L is true of legal traces.

Trace Functions:

The dot (.) and V are each a *trace function*. The dot concatenates traces to form a new trace, and V, when applied to a trace that returns a value, denotes the value returned by that trace.

Domain Names:

The name of any given domain is a *domain name*. Such domains are said to be *named*. The domains we are primarily interested in are parameter domains for procedure calls. An example is the domain of integers.

Domain Constants:

ϕ_δ, where ϕ denotes any member of a named domain δ, is a *domain constant*. These denote members of the named domains, e. g., "1_{int}" denotes the integer 1.

Domain Functions:

ϕ_δ, where ϕ denotes any well-specified function on the members of a named domain δ, is a *domain function*. If ϕ denotes an n-placed function, then ϕ is *n−ary*. If ϕ denotes a function from δ to some domain δ^*, then ϕ is said to be δ^**−valued*. For example, "$+_{int}$" is a 2-ary, integer-valued function.

Domain Relations:

ϕ_δ, where ϕ denotes any well-specified relation on the members of a named domain δ, is a *domain relation* . If ϕ denotes a relation on n elements, it is *n−ary*. For example, "$<_{int}$" denotes *less than* over the integers.

Domain Variables:

a_δ, b_δ, c_δ, ..., where δ is a domain name, are each a *domain variable*. They can be superscripted.

3.1.3. Formation Rules

The following formation rules show how to combine vocabulary elements to form syntax sentences and assertions about traces. These in turn are combined into trace specifications.[12]

Domain Lists:

domain list $\rightarrow$

domain name |

domain list domain name

A domain list containing n domain name occurrences is called an *n place domain list*.

Syntax Sentences:

syntax sentence $\rightarrow$

procedure name: |

procedure name: domain list |

procedure name: --> domain name |

12. The formation rules are given in a variant of Backus Normal Form.

procedure name: domain list --> domain name

When the syntax sentence is of one of the latter two forms, then the procedure name is said to be a δ–*valued function name* where δ is the rightmost domain name in the sentence. A procedure name followed by a domain list of the form $\delta_1, \ldots, \delta_n$, is said to be of *signature* $<\delta_1, \ldots, \delta_n>$. A procedure name not followed by a domain list is said to be of *signature* $\emptyset$.[13] Any domain name that occurs in a syntax sentence is a *parameter domain*. Any parameter domain that occurs to the right of an arrow is a *return value domain*.

Domain Elements:

domain element $\rightarrow$

domain constant |

domain variable

Argument Lists:

argument list $\rightarrow$

domain element |

argument list, **domain element**

An argument list of the form $\alpha_{\delta 1}, \ldots, \alpha_{\delta n}$ is said to be of *signature* $<\delta 1, \ldots, \delta n>$.

Procedure Calls:

procedure call $\rightarrow$

procedure name |

where the procedure name is of *signature* $\emptyset$.

procedure name(**nonempty argument list**)

where the signature of the argument list is a signature of the procedure name.

Trace Expressions:

trace expression $\rightarrow$

empty trace |

trace variable |

procedure call |

trace expression.trace expression

A trace expression that contains neither a trace variable nor a procedure call that has a domain variable for a parameter is called a *trace constant*.

Terms:

term $\rightarrow$

domain element |

trace expression |

V(**trace expression**) **|**

n-ary domain function(**argument list**)

where the function is of the form ϕ_δ, and the argument list is of signature $<\delta_1, \ldots, \delta_n>$ with $\delta_i = \delta$.

Terms have the following *types*. Domain elements of the form α_δ are of type δ. Trace expressions are of type *trace expression*. Terms of the form ϕ(*argument list*) where ϕ is a δ-valued domain function are of type δ. Terms of the form V(*trace expression*) are untyped. Two terms are of *compatible type* if both have the same type, both are untyped, or one is of type δ where δ is a domain name and the other is untyped. A term is *atomic* if it is either a domain element or a trace expression. A term is *constant* if it is

13. Note that a procedure name can have as many signatures as times it appears in a syntax sentence.

either a domain constant or a trace constant.

Variables:

variable $\rightarrow$

  domain variable |

  trace variable

Assertions:

assertion $\rightarrow$

  L(**trace expression**) **|**

  n-ary domain relation(**argument list**) **|**

   where the relation name is of the form R_δ and the argument list is of signature $<\delta_1, \ldots, \delta_n>$ with $\delta_i = \delta$.

  term=**term |**

   where the terms are of compatible type

  $\neg$**assertion |**

  (**assertion** $\&$ **assertion**) **|**

  (**assertion** v **assertion**) **|**

  (**assertion** --> **assertion**) **|**

  (**assertion** <--> **assertion**) **|**

  ($\exists$**variable**)**assertion |**

  (**variable**)**assertion**

For the sake of readability, assertions of the form $\neg\phi=\psi$ will be written as $\phi\neq\psi$, and parentheses will be dropped from the outside of Boolean expressions when no ambiguity results. Each occurrence of a variable ϕ in an assertion $(\phi)\Theta$ or $(\exists\phi)\Theta$ is said to be *bound*. Occurrences that are not bound are *free*. An assertion is *closed* if it contains no free occurrences of any variable.

Definition:

A *trace specification* is an ordered pair (syntax specification, semantic specification), where a *syntax specification* is a recursive set of syntax sentences and a *semantic specification* is a recursive set of assertions.

A trace specification is *proper* if every assertion in its semantic specification is closed. Since all assertions in a proper trace specification are closed, we can abbreviate assertions of the form $(v)\Theta$ by Θ. For the rest of this paper, we will mean by *trace specification* a proper trace specification, unless explicitly stated otherwise.

3.1.4. Language Extensions

For ease of expression, the trace relation symbol $\equiv$ is introduced into the language as an abbreviational device. As mentioned above, the relation holds between two traces if and only if they agree on current and future legality and on all future return values. Its formal definition follows.[14]

Definition:

For any trace expressions ϕ and ψ,

$$
\begin{aligned}
&\phi\equiv\psi =_{def}\\
&\quad (T)((L(\phi.T) <\text{-->} L(\psi.T))\ \&\\
&\qquad (T\neq e\ \text{-->}\\
&\qquad\quad (((\exists v)V(\phi.T)=v\ <\text{-->}(\exists v)V(\psi.T)=v)\ \&\\
&\qquad\quad ((\exists v)V(\phi.T)=v\ \text{-->}\ V(\phi.T)=V(\psi.T)))))
\end{aligned}
$$

14. See [12] for a formalization where $\equiv$ is treated as a language primitive instead of as an abbreviation.

Although trace specifications can contain infinitely many syntax sentences and assertions, we have yet to introduce a notation for writing such long specifications. Such specifications are written using syntax and assertion schemata. A *syntax schema* is a syntax sentence in which at least one occurrence of one or more domain names, say δ, has been replaced either by a string of the form [δ] or by a string of the form $\delta_1 \cdots \delta_n$. In the first case the schema is expanded to two syntax sentences, one that is identical to the schema, but with the brackets deleted, and one that is identical to the schema, but with the brackets and the enclosed parameter deleted. Hence, *POP* : [*int*] expands to *POP* : and *POP* : *int*. In the second case the schema is expanded to form the infinite number of sentences (or schemata) that can be formed by replacing the leftmost occurrence of the dotted string by δ_1, $\delta_1 \delta_2$, etc. in turn. Hence, the sentence *PUSH* : *int* $\cdots$ *int* expands to *PUSH* : *int*, *PUSH* : *int int*, etc. When there is more than one such string in a sentence, the resulting schemata are expanded in turn, leftmost first.

An *assertion schema* is formed by replacing one or more parameters of the same type in a procedure call by a string of the form $v_m, \ldots, v_n$ where m is a positive integer and v is a variable of the same type as the replaced parameter(s). Such a schema is expanded by replacing the leftmost occurrence of such a string by v_m; v_m, v_{m+1}; etc. in turn. If there is more than one such string, the resulting schemata are expanded in turn, leftmost first, except that when there are identical variables, a string cannot go beyond the subscript generated by a string to the left of it. If this makes it impossible to substitute for a string, then the entire call (with its preceding dot or trailing dot, but not both) is deleted from the containing trace. If this renders the containing assertion ill-formed, then the assertion is not substituted for the schema. Hence, $PUSH(x_1, \ldots, x_n) \equiv PUSH(x_1).PUSH(x_2, \ldots, x_n)$ expands to $PUSH(x_1) \equiv PUSH(x_1)$, $PUSH(x_1,x_2) \equiv PUSH(x_1).PUSH(x_2)$, etc.

It should be noted that the schemata described here are only one possible way to extend the formal language. Any extension to or relaxation of the formal language is allowable as long as a method is given for reducing the new constructs into primitive notation.

3.2. A Formal Semantics for Trace Specification

So far we have relied on an intuitive understanding of the meanings of the symbols contained in our trace specification language. However, a formal semantics is necessary if we are to reason rigorously about the structures that can serve as models for a trace specification. Not every structure can serve as such a model. There are implicit restrictions on the denotations that are acceptable for the symbols of our language. For example, the denotation of the trace concatenation symbol must be associative. In this section of the paper, we define a model and define what it means for an assertion to be true in a model. This will support rigorous definitions of the semantic concept of consistency for a trace specification, i. e., having a model, and the semantic concept of totalness for a trace specification, i. e., not having models that differ with respect to the values they yield for a legal trace constant ending in a function call. Although I will further discuss these semantic conceptions of consistency and totalness and the relation between models and implementations later, it should be clear that there is a model that makes $V(T)=a$ true if there is an implementation that returns a when accessed by the series of procedure calls T. Hence, semantically inconsistent specifications have no implementation, and specifications that are total in this semantic sense do not have implementations that differ with respect to observable behavior.

3.2.1. Definition of Trace Model

A *trace model* is an ordered pair (D, I) where D is a disjoint tuple of domains and I is a function from syntactic constructs in **L** to their denotations in D.[15] More specifically, $D = (D_T, D_1, \ldots, D_n, D_{n+1}, \ldots, D_m)$ where D_T can intuitively be regarded as consisting of traces, $D_1 \ldots D_n$ can intuitively be regarded as return value domains, and $D_{n+1} \ldots D_m$ can intuitively be regarded as parameter domains that are not return value domains. D_T contains as a subset D_e, intuitively which consists of the null trace, and traces that are formed from elements of D_T by composing them with a procedure call.[16]

15. The restriction that the parameter domains are disjoint is for simplicity. It is possible to axiomatize a version of specification where parameter domains can be subsets of one another.

16. Ideally, the set of trace denotations is the smallest set closed under composition that contains the null trace and each procedure call. However, this restriction cannot be forced axiomatically in first order logic; infinite compositions of procedure calls must be allowed. See the section of this paper on nonstandard models for further discussion of this point.

Further, D_e must be disjoint from the set of procedure call compositions. If for example, $PUSH(a).POP$'s denotation were in D_e, then POP would be both a functional procedure (since $DEPTH.e$ returns a value) and a nonfunctional procedure (since e does not return a value). More formally, let P be the set of denotations of procedure calls, i. e., $P = Rng(I/\{v : v \text{ is a procedure call}\})$. Our requirement on D_T is that if $x \in D_T$, then $x \in D_e$ if and only if $x \notin \{y : y = I[.](u,w)$ for some $u \in D_T$ and $w \in P\}$. D_T also contains as subsets D_L, intuitively consisting of the legal traces, and D_V, intuitively consisting of those traces that end in a function call. (Hence, $D_e \cap D_V = \emptyset$.) D_V is partitioned into subsets D_{Vi} where each D_{Vi} can intuitively be regarded as consisting of those traces that end in a function call of type δ where $I[\delta]=D_i$. Since the null trace is always legal, D_L contains as a subset D_e.

I must assign equality and the set of legal traces to the equality symbol and L, respectively, and it must assign to V a function from legal traces that return a value of type δ to $I[\delta]$. $I[.]$ must be associative, treat the empty trace as an identity element, and maintain the distinction between traces that return a value of type δ and those that do not. It must also respect the fact that if a module aborts, it cannot recover. Finally, I must assign appropriate denotations to domain elements, domain functions and relations, and to trace expressions. A formal statement of these requirements on I follows.

(1) $I[=] = \{(x,x) : x \in \bigcup D\}$.

(2) $I[L] = D_L$.

(3) $I[V] = f : D_{Vi} \cap D_L \rightarrow D_i, 1 \le i \le n$.

(4) $I[.] = f : D_T \times D_T \rightarrow D_T$ such that for all $x,y,z \in D_T$

- (a) $f(f(x,y),z) = f(x,f(y,z))$,
- (b) $f(x,y) = x$ if $y \in D_e$,
- (c) $f(x,y) = y$ if $x \in D_e$,
- (d) $f(x,y) \in D_T \sim D_{Vi}$ if $y \notin D_{Vi} \cup D_e$,
- (e) $f(x,y) \in D_{Vi}$ if $y \in D_{Vi}$,
- (f) $f(x,y) \in D_T \sim D_L$ if $x \notin D_L$.

(5) $I[domain\ name] \in D \sim D_T$.

(6) $I[\alpha_\delta] \in I[\delta]$ where α_δ is a domain element.

(7) $I[f_\delta] = g : I[\delta]^n \rightarrow I[\delta^*]$ where f_δ is a $n-ary$, $\delta^*-valued$ domain function.

(8) $I[R_\delta] \subseteq I[\delta]^n$ where R_δ is a $n-ary$ domain relation.

(9) $I[trace\ expression]$ is defined by recursion:

- (a) $I[e] \in D_e$.
- (b) $I[\phi] \in D_T$ for any trace variable ϕ.
- (c) $I[procedure\ call] = I[\phi](I[\alpha_1],...,I[\alpha_n])$ where ϕ is the procedure name of the call, and α_i is the ith parameter of the call if there is one. $I[\phi]$ is a function from the set of signatures of ϕ to D^* where $D^* = D_{Vi}$ if the procedure is a functional procedure of type δ and $I[\delta]=D_i$, else $D^* = D_T \sim D_V$.
- (d) $I[T.R] = I[.](I[T],I[R])$ where T and R are any trace expressions.

3.2.2. Definition of Truth in a Model

Given our definition of a model, we must now define what it is for such a model to be a model of a particular trace specification. We do this by defining what it is for an assertion to be *true* in a model $M=(D,I)$. Let T and T^* be any trace expressions, R any relation name, and $t_1, \ldots, t_n$ any domain elements, and t and t^* be any terms.

(1) $L(T)$ is true in M if and only if $I[T] \in I[L]$.

(2) $R(t_1, \ldots, t_n)$ is true in M if and only if $(I[t_1],...,I[t_n]) \in I[R]$.

(3) $t=t^*$ is true in M if and only if $I[t]$ and $I[t^*]$ are both defined and $I^*[t]=I[t^*]$ where $I^*[\phi] = I[\phi]$ if ϕ is a domain element or trace expression, $I^*[\phi] = I[V](I[T])$ if ϕ is of the

form

$V(T)$, and $I^*[\phi] = I[f](I[t_1],...,I[t_n])$ if ϕ is of the form $f(t_1,...,t_1)$.

For any Boolean combination of assertions ϕ and ψ we use Tarski's recursive definition of truth [17]. For quantification over variable v, we use a somewhat simpler definition of truth than Tarski's.

(1) $\neg\phi$ is true in M if and only if ϕ is not true in M.

(2) $\phi \;\&\; \psi$ is true in M if and only if ϕ is true in M and ψ is true in M.

(3) $\phi \vee \psi$ is true in M if and only if ϕ is true in M or ψ is true in M.

(4) ϕ --> ψ is true in M if and only if ψ is true in M or ϕ is not true in M.

(5) ϕ <--> ψ is true in M if and only if both ϕ and ψ are true in M, or neither ϕ nor ψ are true in M.

(6) $(\exists v)\phi$ is true in M if and only if ϕ is true in some interpretation $M^*=(D^*,I^*)$ such that $D=D^*$ and $I=I^*$ except perhaps in what it assigns to v.

(7) $(v)\phi$ is true in M if and only if ϕ is true in every interpretation $M^*=(D^*,I^*)$ such that $D=D^*$ and $I=I^*$ except perhaps in what it assigns to v.

Definition:

M is a *model* for a trace specification S if and only if every assertion in the semantic specification of S is true in M.

Definition:

An assertion A is a *semantic consequence* of a specification S, written $S \models A$, if and only if A is true in every model of S.

Definition:

A specification S is *semantically consistent* if and only if it has a model.

Definition

A specification S is *semantically total* if and only if for every trace constant Φ such that $S \models L(\Phi)$, there is a constant a such that $S \models V(\Phi)=a$.

We have already seen that models can often be converted into programs. The converse is true as well. Given any program P, we say that P suggests a model M with the following domains: D_T consists of all possible compositions of procedure calls of P; D_L consists of those elements of D_T that do not result in error; D_V consists of those elements of D_T that end in a function call; and D_e is the null procedure call. D is $(D_T,D_1,...,D_n)$ where D_i is the *ith* parameter domain of the module under an appropriate ordering. If P suggests a model $M=(D,I)$ for a specification S where I is a function that takes a string in S to its namesake in D, then we say S *specifies* P.

Every implementable specification has a model that can be constructed in this fashion. The converse is false unless the specification has a model that contains only computable functions.[17] It should be noted that the language can be restricted to eliminate such functions. I have chosen not to do so here for the sake of a more elegant theory and to emphasize the distinction, often blurred in the literature, between a model and an implementation. All specifications contained in this paper are implementable.

3.3. A Formal Deductive System for Trace Specification

A formal deductive system permits us to derive assertions from trace specifications in a way that can be verified mechanically. Ideally, it would be possible to derive an assertion from a specification if and only if that assertion was a semantic consequence of that specification. Such a system permits one to reason about trace specifications without appealing to semantic properties and serves as the backbone of the quick implementation system discussed above. Such a deductive system yields a definition of *derivation* and allows us to formalize the syntactic notion of consistency, i. e., not being able to derive $V(T)=d$, $V(T)=d^*$, and $d \neq d^*$ from a specification for any trace constant T and domain constants d and d^*, and the

17. As a counterexample, the interested reader can verify that Gödel's provability predicate [4] is specifiable though not recursive.

syntactic notion of totalness, i. e., that if $L(T.P)$ is derivable from a specification for any trace constant T and variable-free function call P, then $V(T.P)=c$ is derivable for some domain constant c. The primary concern of the rest of this paper is to present a formal deductive system for trace specifications and prove that the system is correct.

The following deductive system has been designed to make derivations relatively easy to construct. Systems that lend themselves more easily to computerized verification of derivations have been constructed from this one by replacing the tautology rule by modus ponens and supplementing the axiom set with a complete set of axioms for sentential calculus as can be found, e. g., in [11]. A deduction theorem for the system is proven, and possible extensions, and modifications to the system are discussed at the end of this section.

3.3.1. Definition of Derivation

Let v be any variable; δ any domain name; $t, t_1, t_2, \cdots$ any terms; C any procedure call; and ϕ, ψ any assertions. $\phi v/t$ is the result of replacing every free occurrence of v in ϕ by t except where such a substitution would result in a bound occurrence for t.[18] A well-formed instance of any of the following schemata is an *axiom*.

1. $(\exists v)v=t$
   where v and t are the same type, t atomic.
2. $(v)(\phi \dashrightarrow \psi) \dashrightarrow (\phi \dashrightarrow (v)\psi)$
   where v is not free in ϕ.
3. $((v)\phi \,\&\, (\exists v)v=t) \dashrightarrow \phi v/t$
   where $\phi v/t$ is well-formed.
4. $v=v$
5. $t=t_1 \dashrightarrow (\phi \longleftrightarrow \psi)$
   where ψ is like ϕ except for possibly having some occurrences of t_1 where ϕ has t.
6. $t=t_1 \dashrightarrow (\exists v_{d1})t=v_{d1}\ [\vee ... \vee\ (\exists v_{dn})t=v_{dn}]$
   for each parameter domain di such that t_1 and v_{di} are of compatible type.
7. $T=T.e$
8. $T=e.T$
9. $L(e)$
10. $L(T.S) \dashrightarrow L(T)$
11. $(\exists S)(\exists v_{11})...(\exists v_{1n})T=S.C_1 \dashrightarrow \neg(\exists R)(\exists v_{21})...(\exists v_{2m})T=R.C_2$
    where C_1 is a δ-valued function call with v_{1i} as the *ith* parameter of the call, and C_2 is a procedure call, with v_{2i} as its *ith* parameter, that is not a δ-valued function call.
12. $((\exists S)((\exists v_{11})...(\exists v_{1n})T=S.C_1 \vee ... \vee (\exists av_{m1})...(\exists v_{mk})T=S.C_m)\ \&$
    $L(T)) \longleftrightarrow (\exists v)V(T)=v$
    where C_i is a call on the *ith* δ-valued function name with a_{ij} as the *jth* parameter of the *ith* call and v is of type δ.
13. $T \neq e \longleftrightarrow$
    $(\exists R)((\exists v_{11}) \cdots (\exists v_{1n})T=R.C_1 \vee \cdots \vee (\exists av_{k1})...(\exists av_{km})T=R.C_k)$
    where C_i is a call on the *ith* procedure of S with a_{ij} as the *jth* parameter of the call.

Each of the following is a *rule of inference*.

TC: (Tautological Consequence) If ϕ is a tautological consequence (as, e. g., determined by a truth table) of a (possibly empty) set of earlier lines in a derivation, then ϕ may be entered as a line in that derivation.

18. The sense of *occurrence* used here is the standard one as used, e. g., in [11], extended to regard trace expressions that occur within other trace expressions as occurring in any assertion that the latter occurs in.

UG: (Universal Generalization) If ϕ appears as an earlier line in a derivation, then one may enter $(v)\phi$ as a line in that derivation.

EI: (Existential Interchange) If ϕ appears as an earlier line in a derivation and ψ is like ϕ except for having one or more occurrences of $(\exists v)$ where ϕ has $\neg(v)\neg$ or vice versa, then one may enter ψ as a line in that derivation.

Definition:

A *derivation* from a trace specification S is a finite sequence of lines, each of which is either (1) an assertion contained in the semantic specification of S, (2) an axiom, or (3) an assertion justified by a rule of inference with the restriction that *UG* is not applied to any variable that occurs free in the semantic specification of S. (It should be noted that the restriction on *UG* is otiose when S is proper.).

Definition:

An assertion ϕ is *derivable* from S, written $S \vdash \phi$, if and only if there is a derivation from S that has ϕ as a last line.

Definition:

A specification S is *syntactically consistent* if and only if there is no assertion of the form $A \ \& \ \neg A$ such that $S \vdash A \ \& \ \neg A$.[19]

Definition:

A specification S is *syntactically total* if and only if for every trace constant Φ ending in a function call, $S \vdash L(\Phi)$ only if there is a constant a such that $S \vdash V(\Phi)=a$.

3.3.2. Deduction Theorem

The following theorem will prove useful in later sections of this paper.

Theorem:

If S is a (possibly nonproper) trace specification whose semantic specification contains the assertion ϕ, then $S \vdash \psi$ only if $S^* \vdash \phi \dashrightarrow \psi$ where $S^*=S\sim\{\phi\}$, i. e. S with ϕ removed from its semantic specification.

Proof: Proof is by induction on the length of derivations. As such, we will assume the theorem for all derivations of length less than n and show that it holds for all derivations of length n whose final line is ψ. There are four possible cases:

(1) If ψ is an axiom, then trivially $S^* \vdash \psi$, from which we can infer that $S^* \vdash \phi \dashrightarrow \psi$ by *TC*.

(2) If ψ was inferred by *TC*, then it is a tautological consequence of earlier lines $L_1,...,L_m$. By hypothesis, $S^* \vdash \phi \dashrightarrow L_1,...,S^* \vdash \phi \dashrightarrow L_m$, from which we may infer that $S^* \vdash \phi \dashrightarrow \psi$ by *TC*.

(3) If ψ was inferred by *UG*, then it is of the form $(v)L$ where L appears as an earlier line. If ϕ is not in S, then $S=S^*$ and by *TC*, $S^* \vdash \phi \dashrightarrow \psi$. If ϕ is in S, then by hypothesis, $S^* \vdash \phi \dashrightarrow L$. Since we could apply *UG* to L, v does not occur free in S (or S^*). Therefore, we can apply *UG* to the conditional, obtaining $S^* \vdash (v)(\phi \dashrightarrow L)$. Further, since ϕ is in S, v does not appear free in ϕ, so we can apply axiom (2) and *TC* to obtain $S^* \vdash \phi \dashrightarrow (v)L$.

(4) If ψ was inferred by *EI* from an earlier line L, then $S^* \vdash \phi \dashrightarrow L$, and we can apply *EI* to obtain $S^* \vdash \phi \dashrightarrow \psi$.

3.3.3. Possible Extensions and Modifications

The above system is minimal in that although I think it correctly represents the trace method as described in [1], some have suggested that it be extended to include further axioms that reflect modifications to the original description of the method. Possible extensions include the addition of stronger axioms for trace equality and axioms that allow for a more radical form of nondeterminism than allowed

19. Since our deductive system contains *TC*, $A \ \& \ \neg A$ is derivable if and only if there is a trace constant T ending in a function call such that $V(T)=x$, $V(T)=y$, and $x \neq y$ are derivable for some x and y.

for here -- i.e., one in which the same implementation may return different values for the same string of function calls at different times.

I chose not to extend this system since it is not clear that any improvement such extensions would make in the specification method is not outweighed by the complexity they would add to the system. For example, the former extension adds little, if anything, to the trace method, yet it requires more restrictions to be placed on $I[.]$ to render, e. g., $I[.](I[w],I[x]) \neq I[.](I[y],I[z])$ if x and z are distinct procedure calls and hence, also necessitates the inclusion of a new subset D_P of D_T consisting of the denotations of procedure calls. By complicating the definition of model, it would make it harder to verify that a specification is consistent. The latter extension would make the method more general, yet it would require either the inclusion of temporal elements in the language, forcing us to give up extensionality and causing completeness problems when we include schemata [13], or the introduction of the membership relation into the language, also causing completeness problems.[20]

3.4. Soundness Theorem

We have both a semantics and a deductive system for the trace specification method, and corresponding conceptions of both consistency and totalness. However, we have yet to bridge the gap between them. The following theorem is fundamental in establishing this bridge.

Theorem:

An assertion A is derivable from a set of assertions S only if it is a semantic consequence of S, i. e., $S \vdash A \Rightarrow S \models A$.

Proof: Given any model M of S, we will prove by induction on the length of A's derivation that A is true in M. Assume that $S \vdash_m A \Rightarrow S \models A$, for all $m < n$ where $S \vdash_m A$ means that A is derivable from S by a derivation of m steps. We must show that $S \vdash_n A \Rightarrow S \models A$. If A is an assertion contained in S, the proof is trivial since, by assumption, M models S. If A is licensed by an axiom or rule of inference, we have the following possible cases:

(1) If A was inferred by axiom (2) or a rule of inference, then its truth follows by standard argument from the induction hypothesis since we have employed the standard definition of truth for all connectives.

(2) If A was inferred by axiom (1), then it is of the form $(\exists v)v=t$ where v and t are of the same type and t is atomic. This assertion is true in all models given the interpretation of such terms and of equality.

(3) If A was inferred by axiom (3), then it is of the form $((v)\phi \ \& \ (\exists v)v=t) \dashrightarrow \phi v/t$ where $\phi v/t$ is well-formed. Its truth follows from the fact that $(\exists v)v=t$ is true if and only if t denotes some object in the domain that is of the same type as v. Given that t is such a term, the axiom reduces to a special case of $(v)\phi \dashrightarrow \phi v/t$, which is true by standard argument.

(4) If A was inferred by axiom (4), then it is of the form $v=v$. The truth of this assertion follows from the definition of equality and the interpretation of variables in the definition of a trace model.

(5) If A was inferred by axiom (5), then it is of the form $t=t^* \dashrightarrow (\phi \longleftrightarrow \psi)$ where ψ is like ϕ except for some possible occurrences of t^*. If $t=t^*$ is not true in M, then A is true in M. If $t=t^*$ is true in M, then $\phi \longleftrightarrow \psi$, and therefore A, is true in M.

(6) If A was inferred by axiom (6), it is of the form $t=t^* \dashrightarrow (\exists v_{d1})t=v_{d1} \vee ... \vee (\exists v_{dn})t=v_{dn}$ where each v_{di} is of compatible type with t^*. Its truth follows from the fact that $t=t^*$ can be true only if t denotes an object in some domain of the appropriate type.

(7) If A was inferred by axiom (7) or (8), then it is of the form $T=T.e$ or $T=e.T$. By definition of $I[.]$, $I[T.e] = I[T] = I[e.T]$ for any T in any sequence.

(8) If A was inferred by axiom (9), then it is of the form $L(e)$. But for any model this is true since $I[e] \in I[L]$.

20. See [12] for further discussion on how to extend the system presented here.

(9) If A was inferred by axiom (10), then it is of the form $L(T.S) \text{-->} L(T)$. This is true on all models by clause (f) of the definition of $I[.]$.

(10) If A was inferred by axiom (11), then it is of the form $(\exists S)(\exists v_{11})...(\exists v_{1n})T{=}S.C_1 \text{-->} \neg(\exists R)(\exists v_{21})...(\exists v_{2m})T{=}R.C_2$ where C_1 is a δ-valued function call with v_{1i} as the *ith* parameter of the call, and C_2 is a procedure call, with v_{2i} as its *ith* parameter, that is not a δ-valued function call. Assume A's antecedent is true. $I[C_1] \in D_{Vi}$ where $I[\delta]{=}D_i$ by the definition of $I[\mathit{procedure\ call}]$. Therefore, $I[T] \in D_{Vi}$ by clause (e) of $I[.]$. If A's consequent were false, we would have two possibilities. If C is a function call of type δ^* where $\delta^* \neq \delta$, then $I[T] \in D_{Vk}$ where $k \neq i$. But this is impossible since the D_{Vi} are disjoint. If C is not a function call, then $I[C] \notin D_V$ by definition of $I[\mathit{procedure\ call}]$. Further, $I[R.C] \notin D_V$ by clause (d) of $I[.]$ since procedure calls are not in D_e. Hence, we would have $I[T]$ in both D_V and $D_T{\sim}D_V$, which is impossible.

(11) If A was inferred by axiom (12), then it is of the form $((\exists S)((\exists v_{11})...(\exists v_{1n})T{=}S.C_1 \vee ... \vee (\exists a v_{m1})...(\exists v_{mk})T{=}S.C_m)\ \&$
$L(T)) \text{<-->} (\exists v)V(T){=}v$, where C_i is a call on the *ith* δ-valued function name with a_{ij} as the *jth* parameter of the *ith* call and v is of type δ. Its truth can be seen by noting that the right hand side is true if and only if $I[T] \in D_{Vi} \cap D_L$. By definition $I[V]$ is defined on all and only legal traces in D_V and takes elements of D_{Vi} to D_i.

(12) If A was inferred by axiom (13), then it is of the form $T \neq e \text{<-->} (\exists S)((\exists a_{11})...(\exists a_{1n})T{=}S.C_1 \vee ... \vee (\exists a_{k1})...(\exists a_{km})T{=}S.C_k)$, where C_i is a call on the *ith* procedure name of S with a_{ij} as the *jth* parameter of the call. Its truth can be seen by noting that an element of D_T is not in D_e, if and only if it is equal to $I[.](u,w)$ for some $u \in D_T$ and $w \in Rng(I/\{v : v$ is a procedure call$\})$ where u may be in D_e.

Corollary:

A trace specification is syntactically consistent if it is semantically consistent.

Proof: If a trace specification is not syntactically consistent, then there is an assertion ϕ, such that $\phi\ \&\ \neg\phi$ is derivable from the specification. By the Soundness Theorem, this latter assertion must be a semantic consequence of any model of the specification. Since the assertion is false in all models, the specification has no models.

3.5. Completeness Theorem

The corollary of the Soundness Theorem offers us a method for proving specifications consistent in either sense of the term, *viz.* finding a model. In this section we prove the deductive system complete *vis–a–vis* the semantics, i. e., that every syntactically consistent specification has a model. This shows the universality of model construction for proving specifications, e. g., consistent, and completes the bridge, began in the previous section, between the syntactic conceptions of consistency and totalness on one hand and their semantic counterparts on the other. The reader should be warned that for the rest of this section I use the term *trace specification* to refer to both proper and non-proper specifications, unless explicitly noted otherwise. Further, I often refer to the semantic specification of some trace specification S, simply as S for the sake of brevity, and I use $S \cup \{A\}$ to refer to the specification that results from adding assertion A to the semantic specification of S.

Theorem:

Every syntactically consistent trace specification has a model.

Proof: We follow a method analogous to that contained in [8], which consists of demonstrating that every specification of a certain type has a model and then demonstrating that every consistent specification can be extended to a specification of that type. The algebraically inclined reader will recognize the type of specification we focus on as being similar to an ultrafilter. This renders the second demonstration analogous to the proof that every set of formulas in a Boolean algebra that satisfies the finite intersection property can be extended to an ultrafilter, and suggests the following definitions:

Definition:

A trace specification is *maximally consistent* if and only if it is syntactically consistent and is such that the addition of any further assertion to its semantic specification would render it syntactically inconsistent.

Definition:

A trace specification is ω–*complete* if and only if for each assertion of the form $(\exists v)A$ in its semantic specification for some variable v, there is an assertion of the form Av/t in its semantic specification for some atomic term t of the same type of v.

The following two facts about maximally consistent trace specifications are needed.

Fact 1: For any maximally consistent trace specification S and any assertion A, either A is in the semantic specification of S or $\neg A$ is in the semantic specification of S.

Proof: If neither A nor $\neg A$ is in S, then $S \cup \{A\} \vdash P \ \& \ \neg P$ and $S \cup \{\neg A\} \vdash P \ \& \ \neg P$ for some assertion P. But then $S \vdash A \dashrightarrow P \ \& \ \neg P$ and $S \vdash \neg A \dashrightarrow P \ \& \ \neg P$ by the Deduction Theorem. But by TC, this implies that S is inconsistent.

Fact 2: S is closed under derivation, i. e., if $S \vdash A$ then $A \in S$.

Proof: If $A \notin S$ then $\neg A \in S$ as proven above, and S would be inconsistent.

The theorem now follows from two lemmas.

Lemma:

Every maximally consistent, ω-complete specification S has a model.

Proof: Order all the domain names $d_1, \ldots, d_m$ such that d_1–d_n name all the return value domains. Consider the set of terms $\{t : (\exists v)v{=}t \in S\}$. Divide the terms of this set into equivalence classes $E_t = \{s : s{=}t \in S\}$. (These are equivalence classes, given that S is closed under derivation, since the reflexivity, transitivity, and symmetry of equality for such terms follow from axioms (4)-(5).) Since there are at most a countable number of terms, we can associate each equivalence class E_t with an unique integer N_t. In each case this integer is the denotation of every term in the equivalence class associated with it. Each atomic term receives a denotation since $t{=}t$ is in S for each such t by axiom (1)

This assignment suggest the following domains and denotations.

$D_T = \{x : x = N_\phi$ for some trace expression $\phi\}$.

$D_i = \{x : x = N_t$ for domain element t of type $d_i\}$.

$D_e = \{N_e\}$.

$D_{Vi} = \{x : x = N_{\phi.C}$ for di-valued function call $C\}$.

$D_L = \{x : x = N_\phi$ and $L(\phi)$ is in $S\}$.

$I[\delta] = D_i$, where δ is the *ith* domain name in the ordering.

$I[\alpha] = N_\alpha$ where α is a domain variable or constant.

$I[f] = \{(N_{t1},...,N_{tn},N_t) : f(t1,...,tn){=}t \in S\}$.

$I[R] = \{(N_{t1},...,N_{tn}) : R(t1,...,tn) \in S\}$.

$I[e] = N_e$.

$I[\phi] = \{(N_{P1},...,N_{Pn},N_{\phi(P1,...,Pn)})$: where $\phi(P_1,...,P_n)$ is a well-formed procedure call$\}$.

$I[L] = D_L$.

$I[=] = \{(x,x) : x = N_t$ for some term $t\}$.

$I[.] = \{(N_T,N_R,N_{T.R})$: T and R are trace expressions$\}$.

$I[V] = \{(N_T,N_t) : N_T \in D_V \cap D_L$ and $V(T){=}t \in S\}$.

We must show that ordered pair $M{=}(D,I)$ we have constructed meets the conditions necessary to be a model. The members of D are disjoint since there are no well-formed assertions of the form $t_1{=}t_2$ where t_1 and t_2 are expressions of noncompatible types. Hence, we only must show that D_T is proper. Assume x is in D_T, we must show that $x \in D_e$ if and only if $x \notin \{y : y = I[.](u,w)$ for some $u \in D_T$ and $w \in Rng(I/\{v : v$ is a procedure call$\})\}$. By construction, $x \in D_T$ if and only if there is a trace expression ϕ such that $x = N_\phi$, and $x \in D_e$ if and only if there is a ψ such that $\phi{=}\psi$ and $\psi{=}e$ are in S. Since S is closed under derivation, axiom (5) implies that $x \in D_e$ if and only if $\phi{=}e$ is in S, and by Fact 1, $x \notin D_e$, if and only if $\phi \neq e$ is in S. Assume that $x \in D_e$, and hence, that $\phi = e$ is in S. If x were also a composition of a trace with a procedure call, then there is a ψ such that $\phi = \psi$ and $\psi = T.C$ are in S where C is a procedure call. But this would render S inconsistent by axiom (13), contrary to hypothesis. If on the other hand $x \notin D_e$, then $\phi \neq e$ is in S. Since S is closed under derivation, axiom (13) implies that $(\exists R)((\exists a_{11})...(\exists a_{1n})\phi = R.C_1$ .. $(\exists a_{k1})...(\exists a_{kn})\phi = R.C_k)$, where each C_i is a procedure call, is in S. Since S is ω-complete, an instance of this assertion is also in S. Further, if none of the disjuncts of this assertion were in S, then each of their negations would be in S by Fact 1, and S, contrary to hypothesis, would be inconsistent. Hence, if $x \notin D_e$, there is an assertion of the form $\phi = R.C$ in S where C is a procedure call, and by construction, $x = I[.](I[R], I[C])$. Analogous arguments show that $D_e \subset D_L$ since $L(e)$ is in S by axiom (9), and that no trace is in two D_{Vi} by axiom (11).

Clauses (1), (2), and (5)-(9) of the definition of trace model are satisfied by construction given that S is a well-formed trace specification. The rest of the definition is satisfied since S is maximally consistent, ω-complete and closed under derivation.

With respect to $I[V]$, note that if $N_T \in D_{Vi} \cap D_L$, then T is in an equivalence class that contains some trace expression $R.C$ where C is a di-valued function call, and some trace expression Q such that $L(Q)$ is in S. But then $T = R.C$ and $T = Q$ are in S, which given axiom (5) and that S is closed under derivation, implies that $L(R.C)$ is in S. Hence, $(\exists v_{di})V(T) = v_{di}$ is in S by axiom (12), which in turn implies that $V(T) = t$ is in S for some domain term of type di since S is ω-complete. Further, there is no term of type dj, $j \neq i$ such that $(\exists v_{dj})V(T) = v_{dj}$ is in S since this would render S inconsistent via axioms (11) and (12). Hence, by construction $I[V]$ takes $I[T]$ to di.

Clause (a) of $I[.]$ is trivial and clauses (b) and (c) follow from axioms (7) and (8). Clause (d) is satisfied since as shown above $N_T \notin D_e$ if and only if $T = R.C$ is in S for some function call C. $N_T \in D_{Vi}$ if and only if $T = R.C$ is in S for some di-valued function call C. Assume that $N_T \notin D_{Vi} \cup D_e$. Then $T = R.C$ is in S for some C that is not a function call of type di. Now, for any Q, $N_{Q.T} \in D_{Vi}$ if and only if $Q.T = W.C^*$ for some di-valued function call C^*. Hence, $N_{Q.T} \in D_{Vi}$ implies that $Q.R.C = W.C^*$ is in S where C^* is a di-valued function call and C is not. But this would render S inconsistent by axiom (11) and is, therefore, impossible. Clause (e) holds since if $T = R.C$ is in S for some di-valued function call C, then for any Q, $Q.T = Q.R.C$ is in S. Finally, Clause (f) follows from axiom (10).

Given that M is a model, we show that it makes true every assertion in S. To this end, define the *order* of an assertion A, $O[A]$, as follows:

If A is of the form $R(t_1,...,t_n)$, $L(T)$, or $t = t^*$, then $O[A] = 1$.

If A is of the form $\neg B$, $B \vee C$, $B \;\&\; C$, $B \dashrightarrow C$, or $B \leftarrow\!\!\dashrightarrow C$, then
$O[A] = max(O[B], O[C]) + 1$.

If A is of the form $(v)B$ or $(\exists v)B$ then $O[A] = O[B] + 1$.

We show that every assertion A in S is true by induction on the order of A. If each formula of order less than n is true if and only if it is in S, we show that each formula A of order n is true if and only if it is in S. This establishes *a fortiori* that every assertion is S is true. There are four possible cases:

(1) If $O[A] = 1$ and A is not of the form $t = t^*$, then A is true in M if and only if A is in S by construction. If A is of the form $t = t^*$, we must show that t (and hence, by symmetry, t^*) has a denotation. There are two cases. If t is atomic, then it has a denotation by construction. If t is of the form $V(T)$ or $f(t_1, \ldots, t_n)$, then $(\exists v_{d1})t = v_{d1} \vee ... \vee (\exists v_{dn})t = v_{dn}$ is in S by axiom (6). But as shown in proving that M is a model, a disjunction can be in S only if one of its disjuncts is in S. Hence, $(\exists v)t = v$ is in S for some v_{di}. Since S is ω-complete, $t = s$ is in S for some atomic term s. Hence, t denotes what s denotes.

(2) If $O[A]$ is greater than 1 and A is a truth functional compound of B and C, then by induction hypothesis, B and C are in S if and only if they are true. Consider the case where A is of the form $B \vee C$. If A is in S, and neither B nor C are in S, then $\neg B$ and $\neg C$ are in S by Fact 1, and S would be inconsistent. Therefore, if A is in S either B is in S or C is in S. Hence, if A is in S, either B or C is true by hypothesis, and therefore, A is true. If A is not in S, then $\neg A$ is in S. Hence, neither B nor C can be in S since S is consistent. Hence, by hypothesis neither B nor C is true, from which it follows that A is not true. The argument for other truth functional compounds is analogous.

(3) If $O[A]$ is greater than 1 and A is of the form $(\exists v)B$, then A is in S only if Bv/t is in S for some atomic term t since S is ω-complete. By the induction hypothesis Bv/t is true which implies that B is true in that model that is identical to the M except for assigning N_t to v. Hence, A is true by definition. If A is not in S then $\neg A$, and hence $(v)\neg B$, is in S. If there were an atomic t such that Bv/t were in S, then we would also have $\neg Bv/t$ in S by axioms (1) and (3). Hence, there is no atomic t such that Bv/t is in S since S is consistent. By induction hypothesis, there is no model of the appropriate type that makes B true since by construction, every element in D is denoted by some atomic t. Therefore, A is not satisfied.

(4) The only other possibility is if A is of the form $(v)B$. If A is in S, then Bv/t must be in S for every atomic term since we can derive Bv/t from A for every such term by axioms (1) and (3). By the induction hypothesis, B is true in all appropriate models since every element in D is denoted by some atomic t. Therefore, A is true. If A is not in S, then $\neg(v)B$ must be in S by Fact 1. Hence, $(\exists v)\neg B$ is in S since S is closed under derivation. But S is ω-complete; therefore, $\neg Bv/t$ is in S for some atomic t. Hence, there is a model in which B is not true, and therefore, A is not true.

Given that every maximally consistent, ω-complete specification has a model, all that is left to prove is that any syntactically consistent trace specification is contained in some maximally consistent, ω-complete specification. This establishes *a fortiori* that every syntactically consistent specification has a model since any model of the extension is a model of the original specification.

Lemma:

Every syntactically consistent specification S can be extended to a maximally consistent, ω-complete specification.

Proof: Enumerate all assertions in the trace specification language so that A_i is the *ith* assertion in the enumeration and build the specification Σ as follows:

Σ_0 = the original specification.

$\Sigma_{i+1} = \Sigma_i$ if Σ_i is syntactically inconsistent when A_{i+1} is added to its semantic specification.

$\Sigma_{i+1} = \Sigma_i \cup \{A_{i+1}\}$ if the resulting specification is syntactically consistent and A_i is not of the form $(\exists v)B$.

$\Sigma_{i+1} = \Sigma_i \cup \{A_{i+1}, Bv/x\}$ where x is the alphabetically first variable of the same type as v that does not appear in $\Sigma_i \cup \{A_i\}$, if $\Sigma_i \cup \{A_{i+1}\}$ is syntactically consistent and A_i is of the form $(\exists v)B$.[21]

$\Sigma = \Sigma_\omega$.

Σ is ω-complete by construction. That Σ is maximal can be seen by noting that if some assertion A_{i+1} is not in Σ, then it is because $\{A_{i+1}\} \cup \Sigma_i$ is inconsistent. But for any assertion A, if $\{A\} \cup \Sigma_i$ is inconsistent, then $\{A\} \cup \Sigma$ must be inconsistent since S_i is a subset of S.

That Σ is syntactically consistent is established if we prove that each Σ_i is syntactically consistent since any derivation of a contradiction can involve at most a finite number of premises and every finite set of assertions contained in Σ is contained in some Σ_i. We establish the syntactic consistency of each Σ_i by induction. Σ_0 is consistent by hypothesis. If Σ_i is consistent, then Σ_{i+1} is obviously consistent if either it is equal to Σ_i or it was formed by the addition of some assertion that could be added consistently to Σ_i. Hence, the only problematic case occurs when we add $(\exists v)B$ to Σ_i since we also add an assertion of the form Bv/x, and it is not obvious that this latter assertion is consistent with the resulting specification. However if $\Sigma_i \cup \{\exists(v)B, Bv/x\}$ is inconsistent, then we can derive $P \ \& \ \neg P$ from this specification for some closed assertion P. But by the Deduction Theorem, this implies that we can derive $\neg Bv/x$ from $\Sigma_i \cup \{(\exists v)B\}$ by using the TC rule of inference. Since by hypothesis, x does not occur free in Σ_i or in

21. If the original specification contains only a finite number of assertions in its semantic specification or if it is specified using any of the schemata described earlier in this paper, then there is always such a variable. If for some reason, we have an infinite specification that contains all the variables of a certain type, we can always generate new variables, e. g., by doubling the indices of each variable in the specification.

$(\exists v)B$, we can generalize and derive $(x)\neg B$ which implies that $\Sigma_i \cup \{(\exists v)B\}$ is not consistent.

This completes the proof of our theorem. Three corollaries follow.

Corollary:

A specification S is syntactically consistent only if it is semantically consistent.

Proof: Immediate given the completeness theorem.

Corollary:

$S \models A$ if and only if $S \vdash A$.

Proof: The implication from right to left follows from the Soundness Theorem. Going from left to right, note that if A is not derivable from S, then $S \cup \{\neg A\}$ is syntactically consistent by the Deduction Theorem. Hence, by the Completeness Theorem it has a model, and therefore, it is not the case that $S \models A$.

Corollary:

S is syntactically total if and only if it is semantically total.

Proof: Immediate given the preceding corollary.

3.5.1. Nonstandard Models

As noted when giving the definition of a trace model, it is impossible to axiomatically force in first order logic every trace expression variable to denote only finite strings of procedure calls. As a consequence, we must allow models that are unintuitive in that they contain infinite traces. This may have bothered some readers, but it can now be seen that the presence of these nonstandard models are of no concern. As the Soundness Theorem demonstrates, assertions that are provable are valid even if we allow these nonstandard models, and in proving the Completeness Theorem we showed how to build for any consistent specification a model that does not contain any nonstandard traces. Hence, an assertion is valid if and only if it is valid in a universe consisting only of "standard" traces.

3.6. Future Research

Future research in the trace method can take various forms. First, the desirability and feasibility of extending the model to allow, e. g., more nondeterminism in specifications, should be explored. Second, software as described in [3] for proving specifications consistent and total and for building quick implementations should be built. A pilot project to develop software support was undertaken at the University of North Carolina and is being continued at the Naval Research Laboratory. NRL is also developing software making it easier to use traces by allowing the specifier to communicate in a English-like language. Third, methods for proving the correctness of implementations and the correctness of programs using modules must be developed.

ACKNOWLEDGEMENTS

The influence of David Parnas' work on this paper is obvious. I also wish to thank Karen Dwyer, Dan Hoffman, Mila Majster, Donald Stanat, and David Weiss for helpful criticism of and earlier draft, and Mark Nixon for useful conversation along the way. Part of the research for this paper was done under NSF grant #1-0-110-3276-XA452.

REFERENCES

1. A W. Bartussek and D. L. Parnas, "Using Traces to Write Abstract Specifications for Software Modules," UNC Report TR 77-012, University of North Carolina, Chapel Hill, N. C. 27514 (1977).
2. C. Chang and H. Keisler, *Model Theory,* North-Holland, Amsterdam (1977).
3. J. Dixon, J. McLean, and D. Parnas, "Rapid Prototyping by Means of Abstract Module Specifications Written as Trace Axioms," *ACM SIGSOFT Engineering Notes* **7** *pp. 45-49 (1982).*
4. K. Gödel, "Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme, I," *Monatshefte für Mathematik und Physik* **37** pp. 179-198 (1931).
5. J. Guttag and J. Horning, "The Algebraic Specification of Abstract Data Types," *Acta Informatica* **10** pp. 27-52 (1978).

6. C. Heitmeyer and J. McLean, "Abstract Requirements Specification: A New Approach and its Application," *IEEE Transactions on Software Engineering* **9** pp. 580-589 (1983).

7. C. Heitmeyer and J. McLean, "An Approach to Describing the Functional Requirements of an Embedded Communications System," NRL Report 8604, Naval Research Laboratory, Washington, D. C. 20375 (1982).

8. L. Henkin, "The Completeness of First Order Functional Calculus," *Journal of Symbolic Logic* **14** pp. 159-166 (1949).

9. L. Henkin, J. Monk, and A. Tarski, *Cylindric Algebras, Part 1*, North-Holland, Amsterdam (1971).

10. B. Liskov and V. Berzins, "An Appraisal of Program Specifications," in *Research Directions in Software Technology,* ed P. Wegner, MIT Press, Cambridge, Massachusetts (1979).

11. B. Mates, *Elementary Logic,* Oxford University Press, New York (1972).

12. J. McLean, "A Formal Foundation For the Trace Method of Software Specification," NRL Report 4874, Naval Research Laboratory, Washington, D. C. 20375 (1982).

13. J. McLean, "A Complete System of Temporal Logic for Specification Schemata", in *Logic of Programs Proceedings 1983,* ed. Dexter Kozen, Springer-Verlag, New York (forthcoming).

14. J. Monk *Introduction to Set Theory,* McGraw-Hill, New York (1969).

15. D. L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules," *Communications of the ACM* **14** pp. 1053-1058 (1972).

16. D. L. Parnas, "The Use of Precise Specifications in the Development of Software," in *Information Processing 77,* ed. B. Gilchrist, North-Holland, New York (1977).

17. A. Tarski, "Der Wahrheitsbegriff in den formalisierten Sprachen," *Studia Philosophica* **1** pp. 261-405 (1936).